Tampere University

Lauri Alho

VISUAL REWARD FOR AUTONOMOUS DRIVING

Faculty of Information
Technology and Communication
Sciences
Bachelor's Thesis
May 2019

ABSTRACT

Lauri Alho: Visual Reward for Autonomous Driving
Bachelor's Thesis
Tampere University
Bachelor's Degree Programme in Information Technology
Examiners: Nataliya Strokina and Wenyan Yang
May 2019

Artificial Intelligence (AI) is seen to show wide adaptation possibilities in many fields, and therefore it is used to solve more and more complex problems. One subfield of it is reinforcement learning, which tries to learn a robot to solve a specified task with a given reward function. The reward function is used to tell the robot, how valuable different actions are in different states.

Defining a reward function for a robot in open spaces can be difficult, and one example of this is teaching a robot to drive a car. In these situations, imitation learning and Inverse Reinforcement Learning (IRL) can offer a solution by turning the problem upside down by creating the reward function from expert demonstrations. These can contain any kind of data that the robot uses to learn the correct policy for the task.

This research studies the possibility to use a visual reward for autonomous driving. Driving simulator Carla is used for creating the training data and running the experiments. Expert demonstrations contain driving videos and control data, and latest research results [1] are used for decreasing the required training data to only a dozen of expert demonstrations.

The experiments showed that a visual reward can be used for autonomous driving, when the task is simple. More research should be done for finding working parameters for longer tasks.

Keywords: visual reward, autonomous driving, reinforcement learning, Carla

PREFACE

This thesis concludes my two-year studying of bachelor's degree at Tampere University. I spent around 320 hours for this research, and I believe that it can be seen in this work. In the end, the research problem turned out to be so inspiring, that I will probably continue this research to the near future.

I would like to thank my supervisors Nataliya Strokina and Wenyan Yang for helping me with this thesis. Without your supervision, this thesis would have never been as good as it is.

Tampere, 10.5.2019

Lauri Alho

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
BCE	Binary Cross Entropy
DNN	Deep Neural Network
IRL	Inverse Reinforcement Learning
ML	Machine Learning
MSE	Mean Squared Error
PG	Policy Gradient
ReLu	Rectified Linear Unit
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
a	an action
s	a state
$\boldsymbol{\theta}$, $\boldsymbol{\theta}_t$	parameter vector of target policy
d'	the number of components of $\boldsymbol{\theta}$
π	policy
$J(\boldsymbol{\theta})$	performance measure for policy π with parameter $\boldsymbol{\theta}$

1. INTRODUCTION

Artificial Intelligence (AI) is currently widely used due to its possibilities in solving problems in many different fields. It is used for example in spam filters, sound synthesizers and predicting house prices. Its adaptation possibilities are increasing at the same time with the complexity of the problems.

AI is branched into many subfields, and Machine Learning (ML) is one of them. It is divided into three fields: supervised learning, unsupervised learning and Reinforcement Learning (RL). [2] The main idea in RL is to let a robot learn from its mistakes. First, a reward is defined for every action, which the robot can take in any state. After that, the robot is let to figure out which action-state pairs give the highest total reward for the task. [3]

RL can be used for autonomous driving, but there are problems in defining a reward function. For example, defining a reward for different driving styles unambiguously can be difficult. Therefore, Inverse Reinforcement Learning (IRL) can be used in these situations, because it releases the hard reward engineering work by learning from correct policy examples. This is done by creating expert demonstrations from the task. These are then used to automatically create the reward function. Expert demonstrations can contain any kind of data, which specifies the current state of the environment.

This research takes the approach from a paper about vision based unsupervised perceptual rewards [1]. It demonstrates the performance for a robotic arm, whereas this research experiments with its applicability for autonomous driving.

Visual reward from a camera is used in a pre-defined route, which is shown in Figure 1. The route starts with driving straight, turning right at the intersection, and then driving straight to the goal.

Figure 1. *Illustration of the driving area, green dot line shows the preferred trajectory for the task. Red colour illustrates the boundaries of the driving area and orange sections, which are signalled as collisions. Crossing either one ends the driving.*

A user drives the car and creates expert demonstrations for this trajectory. Every demonstration contains a video and driving information of it. This approach also aims to reduce the training data by using only a dozen of expert demonstrations that are supposed to give a good estimate of the policy from the very beginning.

This research is done in simulated environment Carla [4], which makes it easy to automatize the creation of expert demonstrations and the training of the robot. It also contains a synchronous mode for simulator-robot communication, which can be used, when the processing power of the computer is not enough for simulating the environment and the robot in real time.

RL is an active learning process in the sense that the robot chooses its actions based on the feedback it gets from the environment. Figure 2 shows the outline of one step in RL, which goes as follows: first, an interpreter sends a starting observation to an agent. Second, the agent predicts an action for the observation, which is sent to the environment. Consequently, the interpreter creates a new observation after this action is made. Also, information of ending the task is interpreted. Finally, this information is sent back to the agent, which saves the observation-action-reward combination for training, and a new

action is predicted from the observation. This loop continues until the car collides with an obstacle, time runs out or the agent reaches the goal.

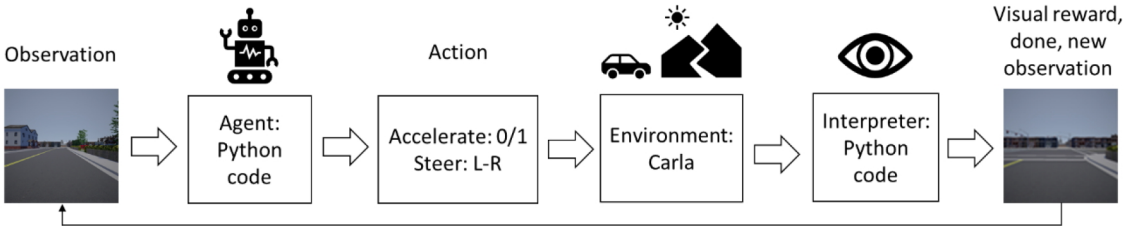

Figure 2. *The outline of one step in Reinforcement Learning.*

Related work is presented in Chapter 2. Implementation starts with recovering the reward from visual feedback in Chapter 3. Chapter 4 shows the model pretraining, which contains the creation of the training data and using pretrained Deep Neural Network (DNN) to train the baseline. Chapter 5 shows the application of a visual reward for autonomous navigation. Chapter 6 describes the experimental environment Carla and the experiments. Chapter 7 contains the conclusions.

The full solution is available at github.com/laurialho/visual-reward.

2. RELATED WORK

Related work contains Inverse Reinforcement Learning and Autonomous Driving with RL.

2.1 Inverse Reinforcement Learning

Imitation learning or learning from demonstration is a growing area in robotics research. It allows to use demonstrations to speed up the learning process and make it safer by limiting the search options. The imitation learning methods either learn the trajectories (known as behaviour cloning) or recover the reward, which reflect the intent of the demonstrator, Inverse Reinforcement Learning (IRL). In the model-based imitation learning the methods access robot dynamics being data-efficient and safe. However, they might be computationally inefficient when learning complicated behaviour.

In IRL the problem is defined by [5] as follows:

> **Given** 1) measurements of an agent's behaviour over time, in a variety of circumstances, 2) measurements of the sensory inputs to that agent; 3) a model of the physical environment (including the agent's body).
>
> **Determine** the reward function that the agent is optimizing.

The reward function is recovered from the demonstrator's policy. However, this policy can be suitable for many different reward functions, and therefore the unique solution cannot be recovered. Different methods can be used to obtain the unique solution. [6] One method is Maximum Margin Planning [7], which is used in [8].

Most of the learning research on autonomous driving is based on carefully engineered reward function. To the best of author's knowledge, very little research can be found on recovering a reward from visual feedback. Many IRL solutions, such as [8], [9], [10] and [11], rely on to distance measure to the surrounding obstacles or goal.

Those, which use vision-based approaches for completing the task, i.e., [12], are based on supervised learning. Perceptual reward functions are used in [13], but the policy is showed with one expert demonstration in simple tasks.

2.2 Autonomous Driving with RL

There are two main approaches in RL: action-value based and Policy Gradient (PG) based methods. Q-learning is an action-value based method, where the reward for each of state-action pair, first, is stored in Q-tables. Fundamental idea in Q-learning is to let the robot fill in the Q-table with as many pairs as possible during training. After each episode, the rewards are updated with discounted reward. When the training is done, the robot can select the most valuable actions in states. [3]

Many RL approaches rely on Q-learning, because it allows discrete modelling of the environment. However, if the environment has a very high number of states and actions, the Q-table size increases and the performance decreases. In these situations, Policy Gradient can be used [3].

In autonomous navigation the reward is typically hand-crafted. Following solutions use Q-learning: In [14], a reward function is learned in Q-learning process, combined with neural network to prevent the generalization of the actions. Environment state contains discrete driving angle and the distance to the goal. The reward is based on the distance between target, collision information and information of finishing the task. In [15], [16] and [17], the task is to navigate to the goal in zone, which contains obstacles. The reward is selected from a discrete value list based on measured distance to the obstacles and the goal.

The other group of methods, Policy Gradient (PG) based approaches, tries to find the most optimal policy for the given task [3]. First group of methods parameterizes the action-state value and the second parametrizes the policy itself. Policy is parameterized usually with a simpler function than action-value table. For example, one can use a Deep Neural Network to parameterize the policy. The model predicts the probabilities of actions at certain step. The higher the probability of the action is, the more probably the higher reward is received from the action. The downside of PG methods is, that they might find only local best policy and not global best policy during optimization. PG is used for autonomous driving for example in [18], [19] and [20].

This research uses a learned visual reward function to teach the driving skill for the robot in the environment. PG-based method was selected for this work since it is more efficient when the demonstrations are provided. It decreases the solution search space significantly and helps to not converge to a local optimum.

3. RECOVERING REWARD FROM VISUAL FEEDBACK

This chapter first presents the algorithm for defining intermediate steps from expert demonstrations, and after that describes the reward prediction from visual features.

3.1 Defining Intermediate Steps

A split algorithm is used to extract abstractly similar parts from expert demonstrations. Figure 3 clarifies the usage of the algorithm, which is adapted from [1] solution.

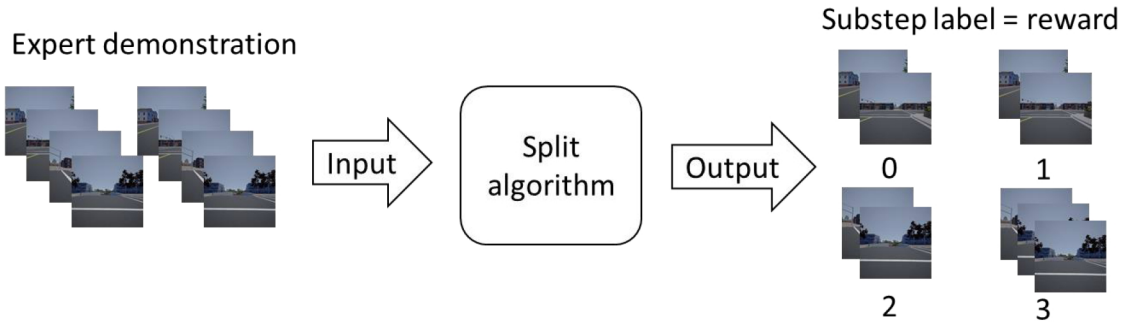

Figure 3. *Split algorithm.*

Program 1 describes the split function. The split function splits one video into smaller parts by returning split points, where substep change. The split points for expert demonstration are calculated by given arguments, which are desired split counts and minimum size of the split. The algorithm runs iteratively and calculates the standard deviation for all greater than minimum sized sections. After that it compares them to find the sections, which have the lowest standard deviation.

The splitting is done to every video, and the power of this algorithm comes from the fact that videos do not have to be the same length, because different length videos still have visually similar parts between other demonstrations.

The complexity of this algorithm is $O(f^n)$ where f is the frame count of the expert demonstration and n is desired split count [1]. Because split algorithm is used only in defining intermediate steps, the complexity of the algorithm does not affect to the performance of on-policy training.

Program 1. *Split function [1]. Join() is a function that combines inputs into one list, AverageStdVideo() is a function that calculates standard deviation over video frames, AverageStdArray() is a function that calculates standard deviation over a list, n is a split count and min_size is a minimum size for a split.*

```
FUNCTION Split(video, start, end, n, min_size, prev_std = []):
  IF n = 1 THEN
    RETURN [], [AverageStdVideo(video[start:end])]
  ENFID
  min_std, min_std_list, min_split <- initialize with empty
  i_start <- start + min_size
  i_last <- end - ((n - 1) * min_size)
  FOR i_start TO i_last DO
    std1 <- AverageStdVideo(video[start:i])
    spits2, std2 = Split(video, i, end, n - 1, min_size, std1 +
prev_std)
    avg_std = AverageStdArray(Join(prev_std, std1, std2))
    IF min_std = empty OR avg_std < min_std THEN
      min_std <- avg_std
      min_std_list <- Join(std1, std2)
      min_split <- Join(i, splits2)
    ENDIF
  ENDFOR
  RETURN min_split, min_std_list
```

After the substeps for the expert demonstrations are created, these can be used as training labels for a DNN. The DNN learns to predict the reward from images.

3.2 Reward Prediction from Visual Features

A DNN is used to predict the reward from the observation. Figure 4 clarifies its functionality.

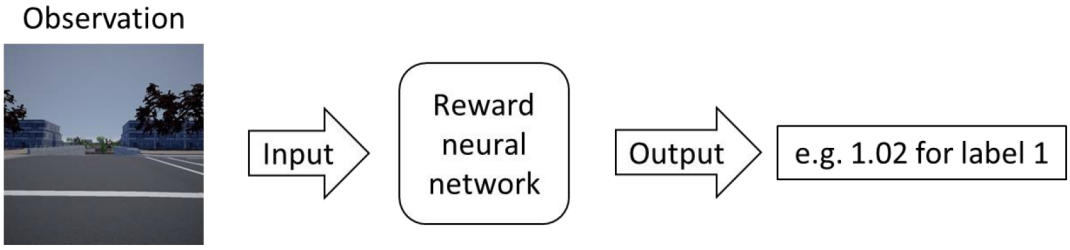

Figure 4. *The functionality of the reward predicting DNN.*

The DNN uses an image as an input and predicts a reward. The structure of the model is shown in Table 1.

Table 1. *The structure of the reward predicting DNN.*

Layer (type)	Output Shape	Param #
image_input (InputLayer)	(None, 299, 299, 3)	0
inceptionv3_imagenet_partial	(None, 35, 35, 192)	173072
flatten (Flatten)	(None, 235200)	0
dense_1 (Dense)	(None, 1)	235201

Total params: 408,273
Trainable params: 235,601
Non-trainable params: 172,672

The model uses first four convolutional layers from InceptionV3 network [21], which is pretrained with ImageNet [22] classifier. The input dimension is 299x299x3 RGB image. Its output is connected to a flatten layer, and the model output is received from a dense layer. The size of the dense layer is one, because linear regression is used for the output, so that the reward can grow linearly even though labels are discrete. Optimizer type can be defined to be stochastic gradient descent (SGD) or Adam, and the loss type is Mean Squared Error (MSE). Reward functions for three different test demonstrations are seen in Figure 5.

The reward is seen to first increase as expected, but with 3 splits it starts to decrease at the end in all demonstrations. The reason for this might be, that the goal section looks similar to the starting point, which makes the reward model to confuse them. With 6 splits this does not happen, and the reward increases almost linearly.

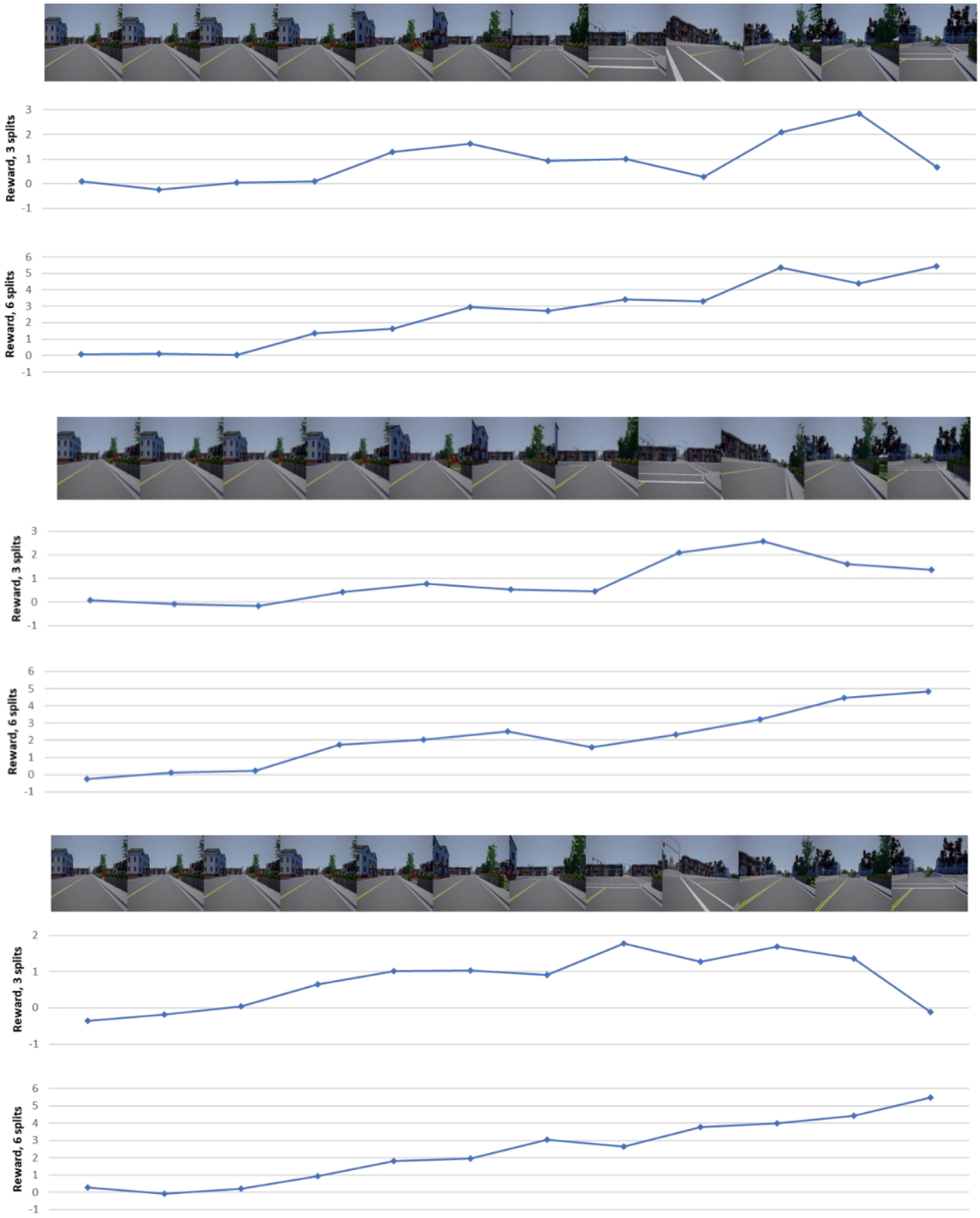

Figure 5. *Reward functions for three demonstration tests. Adam optimizer, learning rate 0.0001, split count 3 and 6.*

4. MODEL PRETRAINING

This chapter shows first the model for action prediction from visual features, and after that the creation of the training data and training the baseline.

4.1 Action Prediction from Visual Features

A DNN is used for predicting actions from observations. The functionality of the DNN is seen in Figure 6.

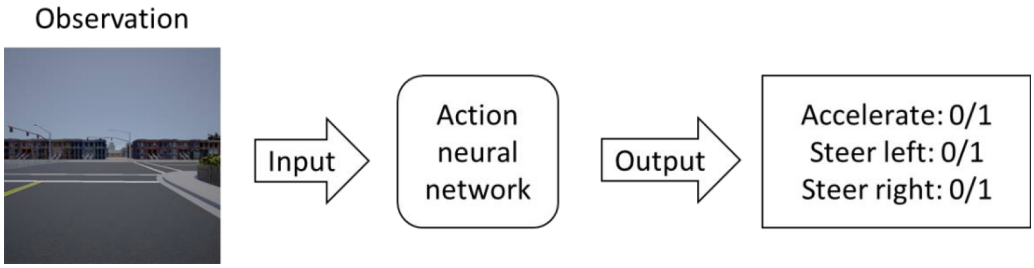

Figure 6. *The functionality of the action predicting DNN.*

The DNN takes an observation as an input and outputs actions between 0 and 1. If prediction for desired action is greater than 0.5, it is considered. Table 2 shows the structure of the action predicting DNN.

Table 2. *The structure of the action predicting DNN.*

Layer (type)	Output Shape	Param #
image_input (InputLayer)	(None, 299, 299, 3)	0
inceptionv3_imagenet_partial	(None, 35, 35, 192)	173072
conv2d_1 (Conv2D)	(None, 35, 35, 32)	153632
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 32)	0
conv2d_2 (Conv2D)	(None, 8, 8, 32)	25632
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 32)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 3)	1539

Total params: 353,875
Trainable params: 181,203
Non-trainable params: 172,672

The DNN contains the trained stripped InceptionV3 model from the reward model, which is frozen after the reward model training. The latter layers contain two convolution layers with 5x5 window size and 32 visual features. Both have a Rectified Linear Unit (ReLU) activation and MaxPooling with 4x4 and 2x2 sized layers respectively after them. The last MaxPooling layer is connected to flatten layer and that correspondingly to dense layer with a sigmoid activation.

The loss function can be defined as a Binary Cross Entropy (BCE) or MSE. When MSE is used, there are two actions instead of three, one for throttling and one for steering. Value 0.5 is defined to mean that steering is not done, whereas greater than that is defined as steering right and lower steering left.

4.2 Creating the Training Data and Finding the Sub Steps

The training data is created in Carla by driving 12 training expert demonstrations and 3 test demonstrations. Observation images are captured from a camera, which is anchored in front of the car. Camera resolution is set to 640x360, and images are converted to 299x299 resolution with bicubic interpolation. Figure 7 shows images of one expert demonstration

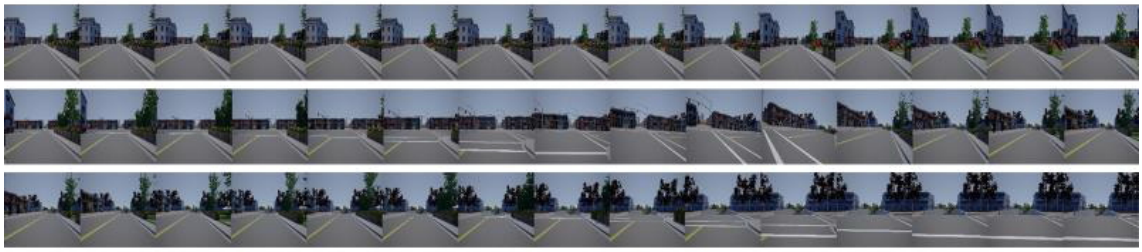

Figure 7. *The images of one expert demonstration.*

This figure shows that the turning part is driven quickly and the taken time for it is approximately 25 %. Correspondingly the first straight section takes around 51 % of the time and last part 24 %.

Figure 8 shows all gathered data of one sample. The driving data contains information of throttling (0 or 1) and steering angle (-0.5 – 0.5) for every observation. Because steering angle is reverted to 0 immediately after the keyboard key is released, any deviation of 0 can be easily transferred to the corresponding steering action.

0.0; 0.4000000059604645

Figure 8. *Data of the sample when the car turns right. Throttle is 0.0, and steering angle is ~0.4.*

The expert demonstrations are inputted into split algorithm one by one, and visually similar sub steps are calculated. The sub step labels for expert demonstrations are then used in supervised learning to train to predict the rewards with the DNN. During training, learning rate must be lower than 0.0001, because higher one results a low accuracy and vast bouncing between the negative gradient.

After the reward DNN is trained, the action DNN is created. It is then pretrained with the expert demonstrations if needed.

4.3 Using Pretrained DNN to Train the Baseline

One way to train the baseline faster is to use a pretrained action DNN to train the baseline. The pretrained DNN can also be compared with other no-pretrained baselines to show performance differences between supervised learning and RL.

Pretraining is done by training the DNN with the expert demonstrations. The expert demonstrations are split into train and test batches, so that the accuracy of the model with unseen data can be evaluated. Maximum number of epochs are selected for pretraining. Pretraining ends automatically if accuracy score increases over 0.999 or value loss is same two times in a row.

After pretraining, the model is used as the baseline, and it can be trained more with the Policy Gradient in Carla. Table 3 shows pretraining with 20 epochs.

Table 3. *Pretraining the action DNN with 20 epochs.*

```
Train on 383 samples, validate on 101 samples
Epoch: 01/20 - loss: 0.4077 - acc: 0.7833 - val_loss: 0.3022 - val_acc: 0.8482
Epoch: 02/20 - loss: 0.2503 - acc: 0.8964 - val_loss: 0.2052 - val_acc: 0.9175
Epoch: 03/20 - loss: 0.1929 - acc: 0.9252 - val_loss: 0.1671 - val_acc: 0.9373
Epoch: 04/20 - loss: 0.1581 - acc: 0.9521 - val_loss: 0.1409 - val_acc: 0.9604
Epoch: 05/20 - loss: 0.1296 - acc: 0.9582 - val_loss: 0.1448 - val_acc: 0.9538
Epoch: 06/20 - loss: 0.1182 - acc: 0.9582 - val_loss: 0.1199 - val_acc: 0.9670
Epoch: 07/20 - loss: 0.1054 - acc: 0.9626 - val_loss: 0.1021 - val_acc: 0.9637
Epoch: 08/20 - loss: 0.0944 - acc: 0.9652 - val_loss: 0.1269 - val_acc: 0.9571
Epoch: 09/20 - loss: 0.0894 - acc: 0.9678 - val_loss: 0.1208 - val_acc: 0.9538
Epoch: 10/20 - loss: 0.0892 - acc: 0.9643 - val_loss: 0.1117 - val_acc: 0.9670
Epoch: 11/20 - loss: 0.0845 - acc: 0.9643 - val_loss: 0.0969 - val_acc: 0.9604
Epoch: 12/20 - loss: 0.0780 - acc: 0.9695 - val_loss: 0.0868 - val_acc: 0.9736
Epoch: 13/20 - loss: 0.0706 - acc: 0.9739 - val_loss: 0.1110 - val_acc: 0.9571
Epoch: 14/20 - loss: 0.0672 - acc: 0.9748 - val_loss: 0.0971 - val_acc: 0.9637
Epoch: 15/20 - loss: 0.0645 - acc: 0.9782 - val_loss: 0.0944 - val_acc: 0.9637
Epoch: 16/20 - loss: 0.0629 - acc: 0.9782 - val_loss: 0.1022 - val_acc: 0.9538
Epoch: 17/20 - loss: 0.0572 - acc: 0.9817 - val_loss: 0.1148 - val_acc: 0.9637
Epoch: 18/20 - loss: 0.0533 - acc: 0.9852 - val_loss: 0.1011 - val_acc: 0.9637
Epoch: 19/20 - loss: 0.0526 - acc: 0.9835 - val_loss: 0.1060 - val_acc: 0.9571
Epoch: 20/20 - loss: 0.0546 - acc: 0.9835 - val_loss: 0.1011 - val_acc: 0.9604
```

5. USING VISUAL REWARD FOR AUTONOMOUS DRIVING

This chapter shows the theory of the Policy Gradient, and how visual rewards are used for autonomous driving.

5.1 Policy Gradient

The theory of this chapter refers to Chapter 13 by [3]. Policy Gradient methods learn a parametrized policy. Policy's parameter vector is $\boldsymbol{\theta} \in \mathbb{R}^{d'}$, where d' represents the number of components of $\boldsymbol{\theta}$. The parametrized policy is written as

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s,\ \boldsymbol{\theta}_t = \boldsymbol{\theta}\}, \tag{5.1}$$

where $\Pr$ means the probability of taking action a in state s with parameter $\boldsymbol{\theta}$ at time t.

The policy parameter $\boldsymbol{\theta}$ is learnt based on the gradient of scalar performance measure, which is defined as $J(\boldsymbol{\theta})$. This can be written as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{5.2}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate, whose expectation approximates the performance measure with $\boldsymbol{\theta}_t$.

5.2 Using Visual Reward with Policy Gradient

A visual reward is used with PG for autonomous driving, so that ideally the correct policy can be learnt inversely. PG is here the gradient of the loss function of the action DNN, which is tried to be minimized. Rewards of the actions are here only used for training the baseline, and therefore they are not directly used during the prediction of the actions. Optimization of the Policy Gradient is described in Program 2.

First, an episode is run on environment. One episode contains multiple steps, which are tuples of observations, predicted actions and rewards from the reward function. After each episode, the baseline is trained with the episode data, so that the actions, which gave a higher reward are selected more often in the future. This is done by calculating the discounted reward for every step and then they are normed if needed. After that, a new episode starts, and this loop continues as long that the defined maximum number of episodes is run, or training is stopped by the user.

Program 2. *Optimizing the Policy Gradient. GetObservation() is a function that gives the latest observation, ChooseAction() is a function that predicts the action, Random() is a function that gives random floating point number between 0 and 1, RandomAction() is a function that gives random action, Append() is a function that appends list, EnvStep() is a function that runs one step in Carla, TrainModel() is a function that trains the model, EnvRestart() is a function that restarts Carla, punish_steps is the amount to punish if steps exceeds, random_action_prob is the floating point number of probability for choosing the random action and gamma is the floating point number between 0 and 1 for discounting the reward.*

```
PROGRAM PG(episodes, steps, punish_steps, random_action_prob, gamma):
  episode <- 0
  done <- False
  observation <- GetObservation()
  WHILE episode != episodes DO
    observations <- []
    actions <- []
    rewards <- []
    step = 0
    episode <- episode + 1
    WHILE True DO
      step <- step + 1
      action <- ChooseAction(observation)
      IF Random() < random_action_prob THEN
        action = RandomAction()
      ENDIF
      observations <- Append(observations, observation)
      actions <- Append (actions,action)
      observation, reward, done <- EnvStep(action)
      rewards <- Append (rewards, reward)
      IF step = steps THEN
        done <- True
        reward <- reward + punish_steps
      ENDIF
      IF done = True THEN
        TrainModel(observations, actions, rewards, gamma)
        BREAK
      ENDIF
    ENDWHILE
    EnvRestart()
  ENDWHILE
```

6. EXPERIMENTS

Experiments are divided into following subsections: Chapter 6.1 describes the experimental environment Carla and Chapter 6.2 used data and parameters for experiments. Chapter 6.3 shows experiments with pretrained baseline and Chapter 6.4 Reinforcement Learning with untrained baseline.

6.1 Experimental Environment: Carla

Figure 9 shows a snapshot of Carla, which is an open-source simulator for autonomous driving research [4]. It is designed to work as a server-client system, which means that one server program runs the environment and one or multiple client programs connect to it. All operations are sent to server through the client program. Carla contains multiple maps with different environments, such as rural areas and cities. Many attributes of the environment can be controlled, such as vehicles, weather, traffic lights and game physics.

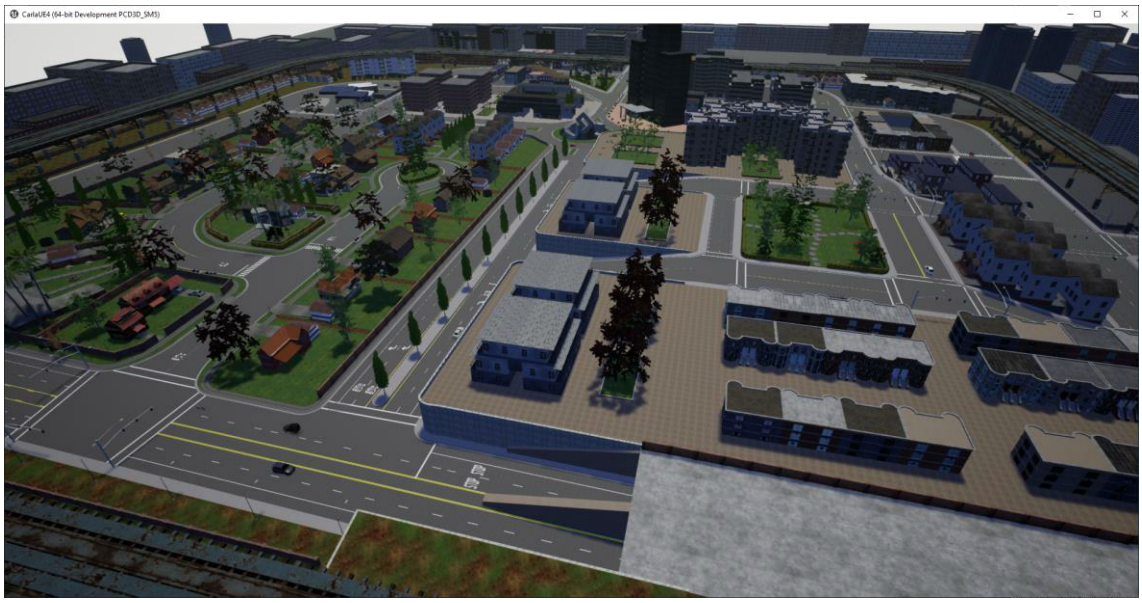

Figure 9. *Carla driving simulator, version 0.94*

Prebuild Carla version 0.94 for Windows is used for these experiments, and it is run on Windows Server 2019 operating system with Intel Core i7-7820X processor, 8 sticks of Kingston HyperX 8 GB DDR4-2666 MHz memory, 2x Asus GTX 1080 Ti graphical processing units and Western Digital WDS512G1X0C 512 GB NVME SSD.

Default settings with following changes are used for the server:

- Map: Town03
- Benchmark: on, steps/frames per second: 60
- Server windowed with 960x960 resolution
- Synchronous mode: on
- Graphical quality: Low

Following starting location is defined for the car on the client side: x = 97.3, y = -129.6, z = 8.0, yaw = 0.0 and roll = 0.0.

6.2 Data and Parameters

Used parameters for split algorithm are splits count 3 or 6 and minimum size of splits as big as the shortest expert demonstration allows. For example, if shortest demonstration is 24 images, with 3 splits the minimum size will be 8 for all the demonstrations. Used parameters for the reward model are listed in Table 4.

Table 4. *Parameters used for the reward model.*

Optimizer	Loss function	Epochs	Learning rate
Adam	MSE	80	0.0001

Automatically gathered data for every step during RL contains the converted image from the camera, the predicted action and the visual reward. Besides, task accomplished status (when the car reaches the goal) and exception status (the car runs out of the driving area) are gathered as well.

Parameters for RL contains gamma value to discount the reward across the episode, the maximum steps allowed per episode and the maximum number of episodes. Probability for random action is also defined, so that the car does actions which the baseline wouldn't otherwise do. The probability for random actions decreases at the same time as ran episode count increases. Following formula is used to calculate the probability for taking a random action:

$$\Pr\{random\ action\} = max\_prob - \frac{episode}{episodes} \tag{6.1}$$

where *max_prob* means the starting probability for random action, *episode* current episode and *episodes* the total number of episodes. If the probability decreases over zero, random actions are not used anymore. The random action is defined to stay on for 20 steps, because lower count would not affect enough for throttling. Random action can be selected to use every 20th step.

For the agent, -100 reward is given as a punishment if it collides with obstacles or runs out of steps. If the agent finishes the task, 100 reward is given.

6.3 Pretrained Baselines

Four pretrained action DNNs are created, two with BCE optimizer and two with MSE. Table 5 shows the used parameters.

Table 5. Parameters used for the action models.

Loss function	Splits	Optimizer	Epochs	Learning rate
BCE	3	Adam	40	0.0001
BCE	6	Adam	40	0.0001
MSE	3	Adam	40	0.0001
MSE	6	Adam	40	0.0001

For MSE the threshold for steering left is set to 0.3333 and right 0.6666. The results after the training are seen in Table 6.

Table 6. *Pretrained BCE and MSE models. 20 episodes run.*

Loss function	Splits	Reached goal (%)
BCE	3	20
BCE	6	90
MSE	3	100
MSE	6	0

The experiments show, that pretrained MSE model with 3 splits can finish the task with a very good accuracy. BCE fails most of the time, because the car steers right too long during the corner, which ends up to collision with the fence. However, with 6 splits results turn upside down. MSE can't reach to the goal whereas BCE can most of the time.

6.4 Experiments with RL Based Methods

Reinforcement Learning is used for baselines with BCE loss function. The results are shown in Table 7.

Table 7. Training results for the baselines, BCE loss function.

Splits	Gamma	Max steps	Start random probability	Train episodes	Reached goal (%)
3	0.2	800	0.95	100	0
3	0.2	800	0.8	1000	0
6	0.2	1000	0.8	100	0
6	0.05	800	0.8	90	0

Common feature for all the tested models is, that they learn to drive straight but fails to steer right in the corner. The 1000 episodes trained model also loses this feature in the end of training and starts to prefer steering slightly right at the beginning, because the received reward from that location is bigger than the reward from driving the wanted trajectory. Therefore, the reward function is a weak point, because it is not trained to give negative reward from those. One alternative solution could be to find suitable values for gamma and punishment, if maximum step count is exceeded.

Another sign of fragility of the reward function was made, when different starting parameters were initially tested. In Figure 1 in Chapter 1 is seen, that the path to the park between the fences is marked as red. This had to be done after some testing, because the robot learned to drive to the park and circle around there. Consequently, the robot received huge rewards at the same time from the reward function, which was not trained to predict correct rewards there.

Multiple parameter combinations are also tested to train more BCE models, which are listed in Table 6. Used parameter combinations are included as a table in Appendix A. To sum up the results, after about 10 training episodes the Policy Gradient RL models starts to prefer actions, which always results to collision or running out of steps. This happens still, even though at first the car is seldomly able to reach the goal. Almost all the tested Policy Gradient RL models ends up to not throttle at all during the start, keeps throttling all the time and not steer, or steers too much right during the corner.

Main reasons for these results might be, that some parts of the route remind too much other parts of the route. This makes the PG to change the behaviour in sub steps which were not intended to change.

7. CONCLUSIONS

Visual reward shows potentiality in autonomous driving, as the Figure 5 in Chapter 3.2 shows. However, it is seen that predicting the visual reward in abstractly similar parts causes problems, because visual rewards are lower or higher than expected. This happens for example, when the car goes off the expert's trajectory. After that, the reward model gives bigger rewards from here than what it would give from the expert's trajectory. One solution would be limiting the maximum reward to the number of splits or training the reward function to give a negative reward in these places.

The reward function seems to contain problems. Many trainings end up into similar situations. In those the Policy Gradient RL model learns to make decisions, which not even try to get to the goal. The reason is the total reward, which is bigger here even with the punishment. If the maximum allowed steps are decreased, the punishment for exceeding it could result the action model to not use any actions anymore. This happens, because almost every action combination ends up into exceeding it. Therefore, increasing the step count and deciding suitable punishments if it is exceeded, could be one solution to increase the robustness of the reward function.

The pretrained models in Chapter 6.3 shows mixed results. The action model with MSE loss and 3 splits can learn the trajectory so well, that it finishes the task every time. However, the model with BCE loss can manage the task with 20 % accuracy, mainly because it learns to oversteer in the corner. More testing with different parameters should be made, so that the accuracy could be increased.

The biggest problem in this research was to find working parameters. The reward function was able to give accurate output for the reward but defining the size of the punishments and maximum steps made problems. If maximum number of steps was too low, the car never learnt to throttle quickly in the beginning to reach goal in time. If maximum steps were increased too much, the robot found a section, where unrealistically high reward was given. Problems were also linked to finding working gamma value. With high gamma value, the punishment was discounted into previous steps, which made the car to stop throttle also in the beginning.

Future researches could investigate the possibilities to use visual reward for autonomous driving by defining more robust reward function and testing more widely different parameters for RL. Also, different split count combinations should be evaluated.

REFERENCES

[1] Sermanet, Pierre; Xu, Kelvin; and Levine, Sergey. 2017. Unsupervised Perceptual Rewards for Imitation Learning. http://arxiv.org/abs/1612.06699.

[2] Raschka, Sebastian; and Vahid, Mirjalili. 2017. Python Machine Learning : Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow. Second edition ed.

[3] Sutton, Richard S.; Barto, Andrew G.; and Bach, Francis. 2018. Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning. Second edition ed. Cambridge, MA; London, England: The MIT Press.

[4] Alexey, Dosovitskiy; Ros, German; Codevilla, Felipe; Lopez, Antonio; and Koltun, Vladlen. 2017. CARLA: An Open Urban Driving Simulator. https://arxiv.org/abs/1711.03938.

[5] Russell, Stuart. 1998. Learning Agents for Uncertain Environments (Extended Abstract). ACM, 1998. doi:10.1145/279943.279964.

[6] Osa, Takayuki; Pajarinen, Joni; Neumann, Gerhard; Bagnell, J. Andrew; Abbeel , Pieter; and Peters, Jan. 2018. An algorithmic perspective on imitation learning. https://arxiv.org/abs/1811.06711.

[7] Ratliff, Nathan D.; Bagnell, J. Andrew; and Zinkevich, Martin A. 2006. Maximum margin planning, Proceedings of the 23rd International Conference on Machine Learning (New York, NY, USA), ICML '06, ACM, 2006, pp. 729–736.

[8] Chen, Xia; and El Kamel, Abdelkader. 2016. Neural Inverse Reinforcement Learning in Autonomous Navigation. Vol. 84.

[9] Beomjoon, Kim; and Pineau, Joelle. 2016. Socially Adaptive Path Planning in Human Environments using Inverse Reinforcement Learning. International Journal of Social Robotics 8 (1): 51-66. doi:10.1007/s12369-015-0310-2.

[10] Choi, Sungjoon; Lee, Kyungjae; Park, Andy; and Oh, Songhwai. 2016. Density Matching Reward Learning. https://arxiv.org/abs/1608.03694.

[11] Sadigh, Dorsa; Landolfi, Nick; Sastry, Shankar S.; Seshia, Sanjit A.; and Dragan, Anca D. Planning for Cars that Coordinate with People: Leveraging Effects on Human Actions for Planning and Active Information Gathering Over Human Internal State. Autonomous Robots 42, no. 7 (2018): 1405-1426.

[12] Darío, Maravall; De Lope, Javier; and Fuentes, Juan Pablo. 2015. Vision-Based Anticipatory Controller for the Autonomous Navigation of an UAV using Artificial Neural Networks. Vol. 151.

[13] Edwards, Ashley; Isbell, Charles; and Takanishi, Atsuo. 2016. Perceptual Reward Functions. https://arxiv.org/pdf/1608.03824.pdf.

[14] L. Niu and L. Li. 2009. Application of Reinforcement Learning in Autonomous Navigation for Virtual Vehicles. doi:10.1109/HIS.2009.118.

[15] Wicaksono, Handy; Khoswanto, Handry; and Kuswadi, Son. 2011. Behaviors Coordination and Learning on Autonomous Navigation of Physical Robot. Telkomnika 9 (3): 473-482.

[16] Wu, Hong-Yan; Liu, Shu-Hua; and Liu, Jie. 2008. A New Navigation Method Based on Reinforcement Learning and Rough Sets. doi:10.1109/ICMLC.2008.4620567.

[17] Fathinezhad, Fatemeh; Derhami, Vali; and Rezaeian, Mehdi. 2016. Supervised Fuzzy Reinforcement Learning for Robot Navigation. Applied Soft Computing 40, 33-41.

[18] C. Wang, J. Wang, Y. Shen, and X. Zhang. 2019. Autonomous Navigation of UAVs in Large-Scale Complex Environments: A Deep Reinforcement Learning Approach. IEEE Transactions on Vehicular Technology 68 (3): 2124-2136. doi:10.1109/TVT.2018.2890773.

[19] Luo, Mincong; Tong, Yin; and Liu, Jiachi. 2019. Orthogonal Policy Gradient and Autonomous Driving Application. doi:10.1109/ICSESS.2018.8663794.

[20] Richard, Liaw; Krishnan, Sanjay; Garg, Animesh; Crankshaw, Daniel; Gonzalez, Joseph E.; and Goldberg, Ken. 2017. Composing Meta-Policies for Autonomous Driving using Hierarchical Deep Reinforcement Learning.

[21] Szegedy, Christian; Vanhoucke, Vincent; Ioffe, Sergey; Shlens, Jonathon; and Wojna, Zbigniew. 2016. Rethinking the inception architecture for computer vision. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[22] J. Deng; W. Dong; R. Socher; L. Li; Kai Li; and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. doi:10.1109/CVPR.2009.5206848.

APPENDIX A

Table 8. *Tested parameter combinations for training more models in Chapter 6.3.*

Sc	Op	Lr	Ep	G	Nr	Rt	Ct	Pc	Ps	Rg
3	Adam	0,0001	40	0,00	0	0	0	0	0	0
3	Adam	0,0001	40	0,10	0	0	0	0	0	0
3	Adam	0,0001	40	0,10	0	0	0	0	0	1000000
3	Adam	0,0001	40	0,10	0	0	0	0	0	1000000
3	Adam	0,0001	40	0,10	1	1	0	-100	-100	100
3	Adam	0,0001	40	0,50	1	1	0	-100	-100	100
3	Adam	0,0001	40	0,95	1	1	0	-10	-100	100
3	Adam	0,0001	40	0,95	1	1	0	-10	-100	100
3	Adam	0,01	100	0,10	0	0	0	-100	-100	100
3	sgd	0,0001	100	0,10	0	0	0	0	0	0
3	sgd	0,001	100	0,10	0	0	0	-100	-100	100
3	sgd	0,01	40	0,10	0	0	0	0	0	0
6	Adam	0,0001	40	0,00	0	0	0	-10	0	0
6	Adam	0,0001	40	0,00	0	0	0	0	0	0
6	Adam	0,0001	40	0,00	0	1	0	0	0	0
6	Adam	0,0001	40	0,00	1	1	0	0	0	0
6	Adam	0,0001	40	0,10	0	0	0	-100	-100	100
6	Adam	0,0001	40	0,95	1	1	0	-10	-100	100
6	Adam	0,0001	40	1,00	0	0	0	-100	-100	100
6	Adam	0,0001	40	1,00	0	1	0	-100	-100	100
6	Adam	0,0001	40	1,00	1	0	0	-100	-100	100
6	Adam	0,0001	40	1,00	1	1	0	-100	-100	100
6	Adam	0,0001	40	1,00	1	1	0	-100	-100	100
6	Adam	0,0001	40	1,00	1	1	1	-100	-100	100
6	Adam	0,0001	40	1,00	1	1	1	-100	-100	100

Legend:

Sc: Splitcount
Op: Action optimizer
Lr: Action learning rate
Ep: Maximum epochs
G: Gamma
Nr: Norm rewards
Rt: Train with rounded actions
Ct: Remove 140 steps from beginning, when trained
Pc: Amount to punish, when collision happens
Ps: Amount to punish, when maximum steps exceeded
Rg: Reward to give, when reaches the goal